DYNAMIC TYPING WITH DEPENDENT TYPES

Xinming Ou, Gang Tan, Yitzhak Mandelbaum and David Walker

Department of Computer Science
Princeton University
{xou,gtan,yitzhakm,dpw}@cs.princeton.edu

Abstract Dependent type systems are promising tools programmers can use to increase the reliability and security of their programs. Unfortunately, dependently-typed programming languages require programmers to annotate their programs with many typing specifications to help guide the type checker. This paper shows how to make the process of programming with dependent types more palatable by defining a language in which programmers have fine-grained control over the trade-off between the number of dependent typing annotations they must place on programs and the degree of compile-time safety. More specifically, certain program fragments are marked *dependent*, in which case the programmer annotates them in detail and a dependent type checker verifies them at compile time. Other fragments are marked *simple*, in which case they may be annotation-free and dependent constraints are verified at run time.

1 Introduction

Dependent type systems are powerful tools that allow programmers to specify and enforce rich data invariants and guarantee that dangerous or unwanted program behaviors never happen. Consequently, dependently-typed programming languages are important tools in global computing environments where users must certify and check deep properties of mobile programs.

While the theory of dependent types has been studied for several decades, researchers have only recently begun to be able to integrate these rich specification mechanisms into modern programming languages. The major stumbling block in this enterprise is how to avoid a design in which programmers must place so many typing annotations on their programs that the dependent types become more trouble than they are worth. In other words, how do we avoid a situation in which programmers spend so much time writing specifications to guide the type checker that they cannot make any progress coding up the computation they wish to execute?

The main solution to this problem has been to explicitly avoid any attempt at full verification of program correctness and to instead focus on verification of safety properties in limited but important domains. Hence, Xi and Pfenning [12] and Zenger [13] have focused on integer reasoning to check the safety of array-based code and also on simple symbolic constraints for checking properties of data types. Similarly, in their language Vault [5], DeLine and Fahndrich use a form of linear type together with dependency to verify properties of state and improve the robustness of Windows device drivers.

These projects have been very successful, but the annotations required by programming languages involving dependent types can still be a burden to programmers, particularly in functional languages, where programmers are accustomed to using complete type reconstruction algorithms. For instance, one set of benchmarks analyzed by Xi and Pfenning indicates that programmers can often expect that 10-20 percent of their code will be typing annotations[1].

In order to encourage programmers to use dependent specifications in their programs, we propose a language design and type system that allows programmers to add dependent specifications to program fragments bit by bit. More specifically, certain program components are marked *dependent*, in which case the type checker verifies statically that the programmer has properly maintained dependent typing annotations. Other portions of the program are marked *simple* and in these sections, programmers are free to write code as they would in any ordinary simply-typed programming language. When control passes between dependent and simple fragments, data flowing from simply-typed code into dependently-typed code is checked dynamically to ensure that the dependent invariants hold.

This strategy allows programmers to employ a pay-as-you-go approach when it comes to using dependent types. For instance, when first prototyping their system, programmers may avoid dependent types since their invariants and code structure may be in greater flux at that time or they simply need to get the project off the ground as quickly as possible. Later, they may add dependent types piece by piece until they are satisfied with the level of static verification. More generally, our strategy allows programmers to achieve better compile-time safety assurance in a gradual and type-safe way.

The main contributions of our paper are the following: First, we formalize a source-level dependently-typed functional language with a syntax-directed type checking algorithm. The language admits programs that freely mix both dependently-typed and simply-typed program fragments.

Second, we formalize the procedure for inserting coercions between higher-order dependently-typed and simply-typed code sections and the generation of intermediate-language programs. In these intermediate-language programs, all dynamic checks are explicit and the code is completely dependently typed. We have proven that the translation always produces wellformed dependently-typed code. In other words, we formalize the first stage of a certifying compiler for our language. Our translation is also total under an admissibility requirement on the dependently-typed interface. Any simply-typed code fragment can be linked with a dependently-typed fragment that satisfies this requirement, and the compiler is able to insert sufficient coercions to guarantee safety at run-time.

[1] Table 1 from Xi and Pfenning [12] shows ratios of total lines of type annotations/lines of code for eight array-based benchmarks to be 50/281, 2/33, 3/37, 10/50, 9/81, 40/200, 10/45 and 3/18.

Finally, we extend our system with references. We ensure that references and dependency interact safely and prove the correctness of the strategy for mixing simply-typed and dependently-typed code. Proof outlines for all our theorems can be found in our companion technical report [9].

2 Language Syntax and Overview

At the core of our system is a dependently-typed lambda calculus with recursive functions, pairs and a set of pre-defined constant symbols. At a minimum, the constants must include booleans **true** and **false** as well as conjunction ($\wedge$), negation ($\neg$),and equality($=$). We use $\lambda x : \tau_1 .\, e$ to denote the function $\mathtt{fix}\, f(x : \tau_1) : \tau_2 . e$ when f does not appear free in e and $\mathtt{let}\ \ x = e_1$ **in** e to denote $(\lambda x : \tau .\, e)\, e_1$.[2]

$$\begin{array}{rcl} \tau & ::= & \tau_b \mid \Pi x : \tau . \tau \mid \tau \times \tau \mid \{x : \tau_b \mid e\} \\ e & ::= & c \mid x \mid \mathtt{fix}\, f(x : \tau_1) : \tau_2 . e \mid e\, e \\ & \mid & \langle e,\, e\rangle \mid \pi_1 e \mid \pi_2 e \mid \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e \end{array}$$

The language of types includes a collection of base types (τ_b), which must include boolean type and unit type, but may also include other types (like integer) that are important for the application under consideration. Function types have the form $\Pi x : \tau_1 . \tau_2$ and x, the function argument, may appear in τ_2. If x does not appear in τ_2, we abbreviate the function type as $\tau_1 \to \tau_2$. Note that unlike much recent work on dependent types for practical programming languages, here x is a valid run-time object rather than a purely compile-time index. The reason for this choice is that the compiler will need to generate run-time tests based on types. If the types contain constraints involving abstract compile-time only indices, generation of the run-time tests may be impossible.

To specify interesting properties of values programmers can use *set types* with the form $\{x : \tau_b \mid e\}$, where e is a boolean term involving x. Intuitively, the type contains all values v with base type τ_b such that $[v/x]e$ is equivalent to **true**. We use $\{e\}$ as a shorthand for the set type $\{x :$ **unit** $\mid e\}$ when x does not appear free in e. The *essential type* of τ, $[\![\tau]\!]$, is defined below.

$$[\![\{x : \tau_b \mid e\}]\!] = \tau_b \qquad [\![\tau]\!] = \tau \ \ (\tau \text{ is not a set type})$$

The type-checking algorithm for our language, like other dependently-typed languages, involves deciding equivalence of expressions that appear in types. Therefore, in order for our type system to be both sound and tractable, we cannot allow just any lambda calculus term to appear inside types. In particular, allowing recursive functions inside types makes equivalence decision undecidable, and allowing effectful operations such as access to mutable storage within types makes the type system unsound. To avoid these difficulties, we categorize a subset of the expressions as *pure terms*. For the purposes of this paper, we limit the pure terms to variables whose essential type is a base type, constants with simple type $\tau_{b_1} \to \cdots \to \tau_{b_n}$, and application of pure terms to pure terms. Only a pure term can appear in a valid type. Note this effectively limits dependent functions to the form $\Pi x : \tau_1 . \tau_2$ where $[\![\tau_1]\!] = \tau_b$[3]. A pure

[2] The typing annotations τ_2 and τ are unnecessary in these cases.

[3] Non-dependent function $\tau_1 \to \tau_2$ can still have arbitrary domain type.

term in our system is also a valid run-time expression, as opposed to a compile-time only object.

As an example of the basic elements of the language, consider the following typing context, which gives types to a collection of operations for manipulating integers (type `int`) and integer vectors (type `intvec`).

```
... -1, 0, 1, ... : int
+, -, * : int -> int -> int
<, <=   : int -> int -> bool
type nat = {x:int | 0 <= x}
length  : intvec -> nat
newvec  : Πn:nat.{v:intvec | length v = n}
sub     : Πi:nat.({v:intvec | i < length v} -> int)
```

The `newvec` takes a natural number `n` and returns a new integer vector whose length is equal to `n`, as specified by the set type. The subscript operation `sub` takes two arguments: a natural number `i` and an integer vector, and returns the component of the vector at index `i`. Its type requires `i` must be within the vector's bound.

Simple and Dependent Typing. To allow programmers to control the precision of the type checker for the language, we add three special commands to the surface language:

$$e ::= \cdots \mid \textbf{simple}\{e\} \mid \textbf{dependent}\{e\} \mid \textbf{assert}(e,\ \tau)$$

Informally, **simple**$\{e\}$ means expression e is only simply well-typed and there is no sufficient annotation for statically verifying all dependent constraints. The type checker must insert dynamic checks to ensure dependent constraints when control passes to a dependent section. For instance, suppose f is a variable that stands for a function defined in a dependently-typed section that requires its argument to have set type $\{x : \texttt{int} \mid x \geq 0\}$. At application site **simple**$\{f\,e\}$ the type checker must verify e is an integer, but may not be able to verify that it is nonnegative. To guarantee run-time safety, the compiler automatically inserts a dynamic check for $e \geq 0$ when it cannot verify this fact statically. At higher types, these simple checks become more general coercions from data of one type to another.

On the other hand, **dependent**$\{e\}$ directs the type checker to verify e is well-typed taking all of the dependent constraints into consideration. If the type checker cannot verify all dependent constraints statically, it fails and alerts the user. We also provide a convenient utility function **assert**$(e,\ \tau)$ that checks at run time that expression e produces a value with type τ.

Together these commands allow users to tightly control the trade-off between the degree of compile-time guarantee and the ease of programming. The fewer **simple** or **assert** commands, the greater the compile-time guarantee, although the greater the burden to the programmer in terms of type annotations. Also, programmers have good control over where potential failures may happen — they can only occur inside a **simple** scope or at an **assert** expression.

For instance, consider the following function that computes dot-product:

```
simple{
  let dotprod = λv1.λv2. let f = fix loop n i sum
                  if (i = n) then sum
                  else loop n (i+1) (sum + (sub i v1) * (sub i v2))
                in f (length v1) 0 0
    in dotprod vec1 vec2 }
```

Function `dotprod` takes two vectors as arguments and returns the sum of multiplication of corresponding components of the vectors. The entire function is defined within a **simple** scope so programmers need not add any typing annotations. However, the cost is that the type checker infers only that `i` is some integer and `v1` and `v2` are integer vectors. Without information concerning the length of the vectors and size of the integer, the checker cannot verify that the `sub` operations are in bound. As a result, the compiler will insert dynamic checks at these points.

As a matter of fact, without these checks the above program would crash if the length of `vec1` is greater than that of `vec2`! To prevent clients of the `dotprod` function from calling it with such illegal arguments, a programmer can give `dotprod` a dependent type while leaving the body of the function simply-typed:

```
dependent {
  let dotprod = λv1:intvec, v2:{v2:intvec | length v1 = length v2}.
                     simple { ... }
   in dotprod vec1 vec2 }
```

The advantage of adding this typing annotation is that the programmer has formally documented the condition for correct use of the `dotprod` function. Now the type checker has to prove that the length of `vec1` is equal to that of `vec2`. If this is not the case the error will be detected at compile time.

Even though the compiler can verify the function is called with valid arguments, it still needs to insert run-time checks for the vector accesses because they are inside a **simple** scope. To add an extra degree of compile-time confidence, the programmer can verify the function body by placing it completely in the **dependent** scope and adding the appropriate loop invariant annotation as shown below.

```
dependent {
 let dotprod = λv1:intvec, v2:{v2:intvec | length v1 = length v2}.
        let f = fix loop (n:{n:nat|n = length(v1)})
                         (i:{i:nat|i <= n}) (sum:int).
                  if (i = n) then sum
                  else loop n (i+1) (sum + (sub i v1) * (sub i v2))
        in f (length v1) 0 0
  in  dotprod vec1 vec2 }
```

With the new typing annotations and some simple integer arithmetic reasoning, our type checker can verify that all the dependent function applications within the function body are well-typed. Once the above code type checks, there can be no failure at run time.

As illustrated by the example, the compiler has the freedom to insert dynamic checks to explicitly verify dependent constraints at run-time. While the kind of run-

$$\frac{\mathcal{F}(c) = \tau}{\Gamma \vdash c : \tau}\ \textit{TConst} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \textit{TVar} \qquad \frac{\Gamma \vdash \tau \text{ valid}}{\Gamma \vdash \textbf{fail} : \tau}\ \textit{TFail}$$

$$\frac{\Gamma \vdash \Pi x : \tau_1.\tau_2 \text{ valid} \qquad \Gamma, f : \Pi x : \tau_1.\tau_2, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fix}\, f(x : \tau_1) : \tau_2.e : \Pi x : \tau_1.\tau_2}\ \textit{TFun}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \Pi x : \tau_1.\tau_2 \\ \Gamma \vdash e_2 : \tau_1 \qquad \Gamma \vdash_{pure} e_2\end{array}}{\Gamma \vdash e_1\, e_2 : [e_2/x]\tau_2}\ \textit{TAppPure} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \\ \Gamma \vdash e_2 : \tau_1\end{array}}{\Gamma \vdash e_1\, e_2 : \tau_2}\ \textit{TAppImPure}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1,\ e_2 \rangle : \tau_1 \times \tau_2}\ \textit{TP} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1 e : \tau_1}\ \textit{TPL} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2 e : \tau_2}\ \textit{TPR}$$

$$\frac{\Gamma \vdash_{pure} e : \textbf{bool} \qquad \Gamma, u : \{e\} \vdash e_1 : \tau \qquad \Gamma, u : \{\neg e\} \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}\ \textit{TIf}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash_{pure} e}{\Gamma \vdash e : \textbf{self}(\tau, e)}\ \textit{TSelf} \qquad \frac{\Gamma \vdash e : \tau' \qquad \Gamma \vdash \tau' \leq \tau}{\Gamma \vdash e : \tau}\ \textit{TSub}$$

Figure 1. Type rules for the internal language

time checks in this example are simple, one has to be careful if the objects passed between dependent and simple sections involve functions, because the dependent constraints may appear at both covariant and contravariant positions. We formalize the process of inserting dynamic checks in the type coercion judgment discussed in the next section.

3 Formal Language Semantics

We give a formal semantics to our language in two main steps. First, we define a type system for our internal dependently-typed language which contains no **dependent**{}, **simple**{} or **assert** commands. Second, we simultaneously define a syntax-directed type system and translation from the surface programming language into the internal language. We have proven that the translation always generates well-typed internal language terms. Since the latter proof is constructive, our translation always generates expressions with sufficient information for an intermediate language type checker to verify type correctness.

Internal Language Typing. The judgment $\Gamma \vdash e : \tau$ presented in Figure 1 defines the type system for the internal language. The context Γ maps variables to types and $\mathcal{F}$ maps constants to their types. Many of the rules are standard so we only highlight a few. First, the **fail** expression, which has not been mentioned before is used to safely terminate programs and may be given any type. Dependent function introduction is standard, but there are two elimination rules. In the first case, the function type may be dependent, so the argument must be a pure term (judged by $\Gamma \vdash_{pure} e$), since only pure terms may appear inside types. In the second case, the argument may be impure so the function must have non-dependent type. When type checking an `if` statement, the primary argument of the `if` must be a pure boolean term and this argument (or its negation) is added to the context when checking each branch[4].

The type system has a *selfification* rule (*TSelf*), which is inspired by dependent type systems developed to reason about modules [7]. The rule applies a "selfification" function, which returns the most precise possible type for the term, its *singleton type*. For instance, though x might have type `int` in the context, **self**$(\texttt{int}, x)$ produces the type $\{y : \texttt{int} \mid y = x\}$, the type of values exactly equal to x. Also, the constant $+$ might have type $\texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int}$, but through selfification, it will be given the more precise type $\Pi x : \texttt{int}.\Pi y : \texttt{int}.\{z : \texttt{int} \mid z = x + y\}$, the type of functions that add their arguments. Without selfification, the type system would be too weak to do any sophisticated reasoning about variables and values. The selfification function is defined below. Notice that the definition is only upon types that a pure term may have.

$$\begin{aligned}
&\textbf{self}(\tau_b, e) = \{x : \tau_b \mid x = e\}\\
&\textbf{self}(\{x : \tau_b \mid e'\}, e) = \{x : \tau_b \mid e' \wedge x = e\}\\
&\textbf{self}(\tau_b \rightarrow \tau, e) = \Pi x : \tau_b.\textbf{self}(\tau, e\, x)
\end{aligned}$$

Finally, the type system includes a notion of subtyping, where all reasoning about dependent constraints occur. The technical report [9] gives the complete subtyping rules. The interesting case is the subtype relation between set types. As stated below, $\{x : \tau_b \mid e_1\}$ is a subtype of $\{x : \tau_b \mid e_2\}$ provided that $e_1 \supset e_2$ is **true** under assumptions in Γ. Term $e_1 \supset e_2$ stands for the implication between two boolean terms.

$$\frac{\Gamma \vdash \{x : \tau_b \mid e_1\}\ \text{valid} \qquad \Gamma \vdash \{x : \tau_b \mid e_2\}\ \text{valid} \qquad \Gamma, x : \tau_b \models e_1 \supset e_2}{\Gamma \vdash \{x : \tau_b \mid e_1\} \leq \{x : \tau_b \mid e_2\}}$$

Here, $\Gamma \models e$ is a logical entailment judgment that infers truth about the application domains. For example it may infer that $n : \texttt{int} \models n \leq n + 1$. We do not want to limit our language to a particular set of application domains so we leave this judgment unspecified but it must obey the axioms of standard classical logic. A precise set of requirements on the logical entailment judgment may be found in the technical report [9].

Surface Language Typing and Translation. We give a formal semantics to the surface language via a type-directed translation into the internal language. The translation has the form $\Gamma \vdash_w e \rightsquigarrow e' : \tau$ where e is a surface language expression and e' is the resulting internal language expression with type τ. w is a type checking mode which is either dep or sim. In mode dep every dependent constraint must be *statically* verified, whereas in mode sim if the type checker cannot infer dependent constraints statically it will generate dynamic checks. It is important to note that this judgment is a syntax-directed function with Γ, w and e as inputs and e' and τ uniquely determined outputs (if the translation succeeds). In other words, the rules in Figure 2 defines the type checking and translation algorithm for the surface language.

[4] $\Gamma \vdash_{pure} e : \tau$ is the same as $\Gamma \vdash_{pure} e$ except that it also returns the simple type of the pure term

$$\frac{\Gamma \vdash_{pure} c \qquad \mathcal{F}(c) = \tau}{\Gamma \vdash_w c \rightsquigarrow c : \textbf{self}(\tau, c)}\ \textit{ATConstSelf} \qquad \frac{\Gamma \nvdash_{pure} c \qquad \mathcal{F}(c) = \tau}{\Gamma \vdash_w c \rightsquigarrow c : \tau}\ \textit{ATConst}$$

$$\frac{\Gamma \vdash_{pure} x \qquad \Gamma(x) = \tau}{\Gamma \vdash_w x \rightsquigarrow x : \textbf{self}(\tau, x)}\ \textit{ATVarSelf} \qquad \frac{\Gamma \nvdash_{pure} x \qquad \Gamma(x) = \tau}{\Gamma \vdash_w x \rightsquigarrow x : \tau}\ \textit{ATVar}$$

$$\frac{\begin{array}{c}\Gamma \vdash \Pi x : \tau_1.\tau_2 \text{ valid} \qquad \Gamma' = \Gamma, f : \Pi x : \tau_1.\tau_2, x : \tau_1 \\ \Gamma' \vdash_w e \rightsquigarrow e' : \tau_2' \qquad \Gamma' \vdash_w e' : \tau_2' \longrightarrow e'' : \tau_2\end{array}}{\Gamma \vdash_w \texttt{fix}\, f(x : \tau_1) : \tau_2.e \rightsquigarrow \texttt{fix}\, f(x : \tau_1) : \tau_2.e'' : \Pi x : \tau_1.\tau_2}\ \textit{ATFun}$$

$$\frac{\begin{array}{c}\Gamma \vdash_w e_1 \rightsquigarrow e_1' : \Pi x : \tau_1.\tau_2 \\ \Gamma \vdash_w e_2 \rightsquigarrow e_2' : \tau_1' \qquad \Gamma \vdash_w e_2' : \tau_1' \longrightarrow e_2'' : \tau_1 \qquad \Gamma \vdash_{pure} e_2''\end{array}}{\Gamma \vdash_w e_1\, e_2 \rightsquigarrow e_1'\, e_2'' : [e_2''/x]\tau_2}\ \textit{ATAppPure}$$

$$\frac{\begin{array}{c}\Gamma \vdash_w e_1 \rightsquigarrow e_1' : \Pi x : \tau_1.\tau_2 \qquad \Gamma \vdash_w e_1' : \Pi x : \tau_1.\tau_2 \longrightarrow e_1'' : \tau_1 \to [\tau_2]_x \\ \Gamma \vdash_w e_2 \rightsquigarrow e_2' : \tau_1' \qquad \Gamma \vdash_w e_2' : \tau_1' \longrightarrow e_2'' : \tau_1 \qquad \Gamma \nvdash_{pure} e_2''\end{array}}{\Gamma \vdash_w e_1\, e_2 \rightsquigarrow e_1''\, e_2'' : [\tau_2]_x}\ \textit{ATAppImPure}$$

$$\frac{\Gamma \vdash_w e_1 \rightsquigarrow e_1' : \tau_1 \qquad \Gamma \vdash_w e_2 \rightsquigarrow e_2' : \tau_2}{\Gamma \vdash_w \langle e_1,\ e_2 \rangle \rightsquigarrow \langle e_1',\ e_2' \rangle : \tau_1 \times \tau_2}\ \textit{ATProd}$$

$$\frac{\Gamma \vdash_w e \rightsquigarrow e' : \tau_1 \times \tau_2}{\Gamma \vdash_w \pi_1 e \rightsquigarrow \pi_1 e' : \tau_1}\ \textit{ATProjL} \qquad \frac{\Gamma \vdash_w e \rightsquigarrow e' : \tau_1 \times \tau_2}{\Gamma \vdash_w \pi_2 e \rightsquigarrow \pi_2 e' : \tau_2}\ \textit{ATProjR}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{pure} e : \textbf{bool} \\ \Gamma, u : \{e\} \vdash_w e_1 \rightsquigarrow e_1' : \tau_1 \qquad \Gamma, u : \{e\} \vdash_w e_1' : \tau_1 \longrightarrow e_1'' : \tau_1 \sqcup \tau_2 \\ \Gamma, u : \{\neg e\} \vdash_w e_2 \rightsquigarrow e_2' : \tau_2 \qquad \Gamma, u : \{\neg e\} \vdash_w e_2' : \tau_2 \longrightarrow e_2'' : \tau_1 \sqcup \tau_2\end{array}}{\Gamma \vdash_w \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow \texttt{if}\ e\ \texttt{then}\ e_1''\ \texttt{else}\ e_2'' : \tau_1 \sqcup \tau_2}\ \textit{ATIfPure}$$

$$\frac{\Gamma \nvdash_{pure} e \qquad \Gamma \vdash_w \texttt{let}\ \ x = e\ \textbf{in}\ \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow e' : \tau}{\Gamma \vdash_w \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow e' : \tau}\ \textit{ATIfImPure}$$

$$\frac{\Gamma \vdash_{dep} e \rightsquigarrow e' : \tau' \qquad \Gamma \vdash \tau \text{ valid} \qquad \Gamma \vdash_{sim} e' : \tau' \longrightarrow e'' : \tau}{\Gamma \vdash_{dep} \textbf{assert}(e,\ \tau) \rightsquigarrow e'' : \tau}\ \textit{ATAssert}$$

$$\frac{\Gamma \vdash_{sim} e \rightsquigarrow e' : \tau}{\Gamma \vdash_{dep} \textbf{simple}\{e\} \rightsquigarrow e' : \tau}\ \textit{ATDynamic} \qquad \frac{\Gamma \vdash_{dep} e \rightsquigarrow e' : \tau}{\Gamma \vdash_{sim} \textbf{dependent}\{e\} \rightsquigarrow e' : \tau}\ \textit{ATStatic}$$

Figure 2. Surface language type checking and translation

Constants and variables are given singleton types if they are pure via the selfification function (*ATConstSelf* and *ATVarSelf*), but they are given less precise types otherwise (*ATConst* and *ATVar*). To translate a function definition (*ATFun*), the function body e is first translated into e' with type τ_2'. Since this type may not match the annotated result type τ_2, the *type coercion judgment* is called to coerce e' to τ_2, possibly inserting run-time checks if the type checking mode is sim.

The type coercion judgment has the form $\Gamma \vdash_w e : \tau \longrightarrow e' : \tau'$. It is a function, which given type checking mode w, context Γ, expression e with type τ, and a target type τ', generates a new expression e' with type τ'. The output expression is equivalent to the input expression aside from the possible presence of run-time checks. We will discuss the details of this judgment in a moment.

There are two function application rules, distinguished based on whether the argument expression is judged pure or not. If it is pure, rule *ATAppPure* applies and the argument expression is substituted into the result type. If the argument expression is impure, rule *ATAppImpure* first coerces the function expression that has a potentially dependent type $\Pi x : \tau_1 . \tau_2$, to an expression that has a non-dependent function type $\tau_1 \rightarrow [\tau_2]_x$. $[\tau]_x$ returns the type with all occurrences of variable x removed. It is defined on set types as follows and recursively defined according to the type structures for the other types.

$$[\{y : \tau_b \mid e\}]_x = \tau_b \ \ (x \in FV(e))$$
$$[\{y : \tau_b \mid e\}]_x = \{y : \tau_b \mid e\} \ \ (x \notin FV(e))$$

Note that in both application rules the argument expression's type τ_1' may not match the function's argument type so it is coerced to an expression e_2'' with the right type.

In type checking an `if` expression, the two branches may be given different types. So they are coerced to a common type $\tau_1 \sqcup \tau_2$ (*ATIfPure*). Informally, $\tau_1 \sqcup \tau_2$ recursively applies disjunction operation on boolean expressions in set types that appear in covariant positions and applies conjunction operation on those on contravariant positions. For example,

$$\{x : \texttt{int} \mid x < 3\} \sqcup \{x : \texttt{int} \mid x > 10\} = \{x : \texttt{int} \mid x < 3 \vee x > 10\}$$

and

$$\begin{aligned} &(\{x : \texttt{int} \mid x > 3\} \rightarrow \texttt{int}) \sqcup (\{x : \texttt{int} \mid x < 10\} \rightarrow \texttt{int}) \\ = &\ \{x : \texttt{int} \mid x > 3 \wedge x < 10\} \rightarrow \texttt{int} \end{aligned}$$

The precise definition for $\tau_1 \sqcup \tau_2$ can be found in the technical report [9].

The rules for checking and translating **dependent**$\{e\}$ and **simple**$\{e\}$ expressions simply switch the type checking mode from sim to dep and vice versa. The rule for **assert**$(e,\ \tau)$ uses the type coercion judgment to coerce expression e to type τ. Note that the coercion is called with *sim* mode to allow insertion of run-time checks.

Type coercion judgment. The complete rules for the type coercion judgment can be found in Figure 3. When the source type is a subtype of the target type, no conversion is necessary (*CSub*). The remaining coercion rules implicitly assume the subtype relation does not hold, hence dynamic checks must be inserted at appropriate places. Note that those rules require the checking mode be sim; when called with mode dep the coercion judgment is just the subtyping judgment and the type checker is designed to signal a compile-time error when it cannot statically prove the source is a subtype of the target.

$$\frac{\Gamma \vdash \tau \leq \tau'}{\Gamma \vdash_w e : \tau \longrightarrow e : \tau'}\ \textit{CSub}$$

$$\frac{\tau = \tau_b \text{ or } \tau = \{x : \tau_b \mid e_1'\}}{\Gamma \vdash_{sim} e : \tau \longrightarrow \texttt{let}\ \ x = e \ \textbf{in} \ \texttt{if}\ e_1 \ \texttt{then}\ x \ \texttt{else}\ \textbf{fail} : \{x : \tau_b \mid e_1\}}\ \textit{CBase}$$

$$\frac{\Gamma \vdash \tau_1' \leq \tau_1 \qquad \Gamma, y : \Pi x : \tau_1.\tau_2, x : \tau_1' \vdash_{sim} y\, x : \tau_2 \longrightarrow e_b : \tau_2'}{\Gamma \vdash_{sim} e : \Pi x : \tau_1.\tau_2 \longrightarrow (\texttt{let}\ \ y = e \ \textbf{in} \ \lambda x : \tau_1'.\ e_b) : \Pi x : \tau_1'.\tau_2'}\ \textit{CFunCo}$$

$$\frac{\begin{array}{c}\Gamma \not\vdash \tau_1' \leq \tau_1 \qquad \Gamma, x : \tau_1' \vdash_{sim} x : \tau_1' \longrightarrow e_x : \tau_1 \\ \Gamma, y : \tau_1 \rightarrow \tau_2, x : \tau_1' \vdash_{sim} y\, e_x : \tau_2 \longrightarrow e_b : \tau_2'\end{array}}{\Gamma \vdash_{sim} e : (\tau_1 \rightarrow \tau_2) \longrightarrow (\texttt{let}\ \ y = e \ \textbf{in} \ \lambda x : \tau_1'.\ e_b) : (\tau_1' \rightarrow \tau_2')}\ \textit{CFunContNonDep}$$

$$\frac{\begin{array}{c}\Gamma \not\vdash \tau_1' \leq \tau_1 \qquad \tau_1 = \{x : \tau_b \mid e_1\} \qquad \tau_1' = \{x : \tau_b \mid e_1'\} \text{ or } \tau_b \\ \Gamma, y : \Pi x : \tau_1.\tau_2, x : \tau_1 \vdash_{sim} y\, x : \tau_2 \longrightarrow e_b : \tau_2' \\ e_b' = \texttt{if}\ e_1 \ \texttt{then}\ e_b \ \texttt{else}\ \textbf{fail}\end{array}}{\Gamma \vdash_{sim} e : \Pi x : \tau_1.\tau_2 \longrightarrow (\texttt{let}\ \ y = e \ \textbf{in} \ \lambda x : \tau_1'.\ e_b') : \Pi x : \tau_1'.\tau_2'}\ \textit{CFunContDep}$$

$$\frac{\begin{array}{c}\Gamma, y : \tau_1 \times \tau_2 \vdash_{sim} \pi_1 y : \tau_1 \longrightarrow e_1' : \tau_1' \\ \Gamma, y : \tau_1 \times \tau_2 \vdash_{sim} \pi_2 y : \tau_2 \longrightarrow e_2' : \tau_2'\end{array}}{\Gamma \vdash_{sim} e : \tau_1 \times \tau_2 \longrightarrow (\texttt{let}\ \ y = e \ \textbf{in} \ \langle e_1',\ e_2' \rangle) : \tau_1' \times \tau_2'}\ \textit{CPair}$$

Figure 3. Type coercion

Coercion for the base-type case (*CBase*) is straightforward. An `if` expression ensures that the invariant expressed by the target set type holds. Otherwise a runtime failure will occur. With the help of the logical entailment judgment, our type system is able to infer that the resulting `if` expression has the set type.

In general, one cannot directly check at run-time that a function's code precisely obeys some behavioral specification expressed by a dependent type. What we can do is ensure that every time the function is called, the function's argument meets the dependent type's requirement, and its body produces a value that satisfies the promised result type. This strategy is sufficient for ensuring run-time safety. The coercion rules for functions are designed to coerce a function from one type to a function with another type, deferring checks on arguments and results until the function is called.

There are three coercion rules for function types. In all cases the expression that generates the function is evaluated first to preserve the order of effects. Next a new function is constructed with checks on argument and result inserted when necessary. In the case where the new argument type is a subtype of the old one (*CFunCo*), we only need to convert the function body to the appropriate result type. Otherwise checks must be inserted to make sure the argument has the type the old function expects. This can be done by recursively calling the coercion judgment on the argument x to convert it to a term e_x with type τ_1. When the function's type is not dependent, it can receive e_x as an argument (*CFunContNonDep*). But when it is a dependent function, it cannot receive e_x as an argument since e_x contains dynamic checks and is impure[5]. Consequently rule *CFunContDep* uses an `if` statement to directly check the constraint on the dependent argument x. This is possible because x must be a pure term and hence has a base type. If the check succeeds, x is directly passed to the function. For all the three cases, our type system is able to prove the resulting expression has the target function type.

4 Mutable References

The addition of mutable references to our language presents a significant challenge. When sharing a reference between simple and dependent code, it is natural to wish to assign the reference a simple type in the simple code and a dependent type in the dependent code, for example `int ref` and `{x:int|x >= 0} ref`. However, the inequivalence of these types can lead to unsoundness. Therefore, in our surface language, we define two classes of references, τ **ref** and τ **dref**. The former is invariant in its typing, thereby disallowing the transfer of such references between one piece of code and another unless the supplied and assumed types are equal. The latter is more flexible in its typing, but is dynamically checked according to the following two principles: First, the recipient of such a reference is responsible for writing data that maintains the invariants of the reference's donor. Second, the recipient must protect itself by ensuring that data it reads indeed respects its own invariants.

In the internal language, the τ **dref** is implemented as a pair of functions:

$$(\textbf{unit} \rightarrow \tau) \times (\tau \rightarrow \textbf{unit})$$

Intuitively, the first function reads an underlying reference and coerces the value to the right type; the second one coerces the input value to the type of the underlying reference and writes the coerced value into it.

We define a type translation $(\!|\tau|\!)$ to translate surface language types to internal language types. It recursively traverses the type structure of τ and translates any appearance of dynamic references as shown above.

The coercion rules for references allow translation from an expression of τ **ref** to an expression of τ' **dref**, or from τ **dref** to τ' **dref**. But there is no coercion rule from τ **dref** to τ' **ref**, because an expression with τ **dref** will potentially incur runtime failures, while an expression with type τ **ref** will not. Further details of our solution can be found in our companion technical report [9].

5 Language Properties

In this section, we present theorems that state formal properties of our language. We leave details of the proofs and precise definitions to the technical report [9]. First, we proved type safety for the internal language based on a standard dynamic semantics with mutable references:

[5] We also cannot simply write `let` $z = e_x$ **in** $y\,z$ since the effects in e_x do not allow the type system to maintain the proper dependency between x and z in this case.

Theorem 1 (Type safety) *If* $\bullet \vdash e : \tau$*, then* e *won't get stuck in evaluation.*

The proof is by induction on the length of execution sequence, using standard progress and preservation theorems.

The soundness of the type-directed translation for the surface language is formalized as the following theorem.

Theorem 2 (Soundness of translation of surface language) *If* $\Gamma \vdash_w e \leadsto e' : \tau$*, then* $(\!|\Gamma|\!) \vdash (\!|e'|\!) : (\!|\tau|\!)$.

$(\!|e|\!)$ is the expression with every type τ appearing in it replaced by $(\!|\tau|\!)$, and $\forall x \in dom(\Gamma).(\!|\Gamma|\!)(x) = (\!|\Gamma(x)|\!)$.

For all source programs that are simply well-typed (judged by $\Gamma \vdash_0 e : \tau$), if the dependent interface Γ satisfies an admissibility requirement *co_ref*(Γ), the translation is total in *sim* mode:

Theorem 3 (Completeness of translation) *Assuming co_ref*(Γ) *and co_ref*$(\mathcal{F})$*, if* $\Gamma \vdash_0 e : \tau$*, then there exist* e' *and* τ' *such that* $\Gamma \vdash_{sim} e \leadsto e' : \tau'$.

Informally, *co_ref*(Γ) states that in Γ, unchecked reference type (τ **ref**) can only appear in covariant positions. The reason for this restriction is that we cannot coerce a checked reference (τ **dref**) to an unchecked one.

6 Related Work

In this paper, we have shown how to include fragments of *simply-typed* code within the context of a *dependently-typed* language. In the past, many researchers have examined techniques for including *uni-typed* code (code with one type such as Scheme code) within the context of a *simply-typed* language by means of soft typing ([3, 2, 4]). Soft typing infers simple or polymorphic types for programs but not general dependent types.

Necula et al. [8] have developed a soft typing system for C, with the goal of ensuring that C programs do not contain memory errors. Necula et al. focus on the problem of inferring the status of C pointers in the presence of casts and pointer arithmetic, which are either *safe* (well-typed and requiring no checks), *seq* (well-typed and requiring bounds checking) or *dynamic* (about which nothing is known). In contrast, we always know the simple type of an object that is pointed to, but may not know about its dependent refinements.

When dependent types mix with references, one has to be very careful to ensure the system remains sound. Xi and Pfenning [12] shows how to maintain soundness by using singleton types, and restricting the language of indices that appear in the singleton types. Our approach is similar in that we have designated a subset of terms as pure terms, but different in that we accommodate true dependent types. However, the distinction is minor, and the main contribution of this work is the interaction between the dependently-typed world and the simply-typed world.

Walker [11] shows how to compile a simply-typed lambda calculus into a dependently-typed intermediate language that enforces safety policies specified by simple state machines. However, he does not consider mixing a general dependently-typed language with a simply-typed language or the problems concerning mutable references.

In earlier work, Abadi et al. [1] showed how to add a special *type dynamic* to represent values of completely unknown type and a typecase operation to the simply-typed lambda calculus. Abadi et al. use type dynamic when the simple static type of data is unknown, such as when accessing objects from persistent storage or exchanging data with other programs. Thatte [10] demonstrates how to relieve the programmer from having to explicitly write Abadi et al.'s typecase operations themselves by having the compiler automatically insert them as we do. In contrast to our work, Thatte does not consider dependent types or how to instrument programs with mutable references.

In contract checking systems such as Findler and Felleisen's work [6], programmers can place assertions at well-defined program points, such as procedure entries and exits. Findler and Felleisen have specifically looked at how to enforce properties of higher-order code dynamically by wrapping functions to verify function inputs conform to function expectations and function outputs satisfy promised invariants. Our strategy for handling higher-order code is similar. However, Finder and Felleisen's contracts enforce all properties dynamically whereas we show how to blend dynamic mechanisms with static verification.

Acknowledgments. We are grateful to Daniel Wang for his comments on an earlier version of this work. ARDA grant NBCHC030106, and NSF grants CCR-0238328, and CCR-0306313 have provided support for this research. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of ARDA or the NSF.

References

[1] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[2] Alex Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 163–173, January 1994.

[3] R. Cartwright and M. Fagan. Soft typing. In *ACM Conference on Programming Language Design and Implementation*, pages 278–292, 1991.

[4] R. Cartwright and M. Fagan. A practical soft type system for Scheme. *ACM transactions on programming languages and systems*, 19(1):87–152, January 1997.

[5] Rob Deline and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, Utah, June 2001. ACM Press.

[6] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ACM International Conference on Functional Programming*, pages 48–59, Pittsburgh, October 2002. ACM Press.

[7] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 123–137, Portland, OR, January 1994.

[8] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code. In *ACM Symposium on Principles of Programming Languages*, London, January 2002. ACM Press.

[9] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. Technical Report TR-695-04, Department of Computer Science, Princeton University, 2004.

[10] S. Thatte. Quasi-static typing. In *Seventeenth ACM Symposium on Principles of Programming Languages*, pages 367–381, January 1990.

[11] David Walker. A type system for expressive security policies. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 254–267, Boston, January 2000.

[12] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *ACM Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.

[13] Christoph Zenger. Indexed types. In *Theoretical Computer Science*, volume 187, pages 147–165. Elsevier, November 1997.